CSE 582 Autumn 2002 Exam 11/26/02

Name _____

There are **8** questions worth a total of **100** points. Please budget your time so you get to all of the questions. ***Keep your answers brief and to the point.***

You may refer to the following reference materials:

Course lecture slides & notes
Your primary compiler textbook (presumably Cooper & Torczon, Appel, or the dragon book)

No other books or other materials.

Please wait to turn the page until you are told to begin.

CSE 582 Autumn 2002 Exam 11/26/02

Score _____

____	____	____	____	____	____	____	____
1	2	3	4	5	6	7	8

(Remainder of this page intentionally left blank. Questions start on the next page.)

CSE 582 Autumn 2002 Exam 11/26/02

Question 1. (10 points) Regular expressions.
Describe the set of strings that are generated by the each of the following regular expressions.

a) (a | (bc)* d)+

b) ((0 | 1)* (2 | 3)+) | 0011

Question 2. (10 points) Draw a deterministic finite automata (DFA) that recognizes strings generated by (a | (bc)* d)+. You do not need to use the formal algorithms for converting regular expressions to a NFA then a DFA (although those algorithms might provide some insight into what's needed). Just draw a suitable diagram.

Question 3. (10 points) A new programming language is being designed that includes decimal numbers with no exponent. A decimal number has a period to separate the integer and fractional parts of the number and **must** have at least one digit both before and after the decimal point. Furthermore, a decimal number **must not** have any superfluous leading or trailing 0's. Some examples of *legal* decimal numbers are 3.14, 0.01, 17.0, 0.0, 123.00321. Some examples of *illegal* decimal numbers are 00.0, .5, 17, 17., 003.01, 0.0012300 (this last example is illegal because of the trailing 0's; the 0 before the decimal point is required).

Write a regular expression (or collection of regular expressions) to generate legal decimal numbers.

Question 4. (10 points) Suppose we have the following fragment of a Java program (including some language operators that are not included in JFlat)

```
while (xyzzy >= thing) // repeat until thing is big enough
    { x += y;
      thing++; }
```

List in order the tokens that would be produced by a scanner when reading this input. (Use any reasonable set of token names; just be sure they are clear and descriptive.)

CSE 582 Autumn 2002 Exam 11/26/02

Question 5. (12 points) The standard programming language grammar for statements is ambiguous because of the dangling else problem.

stmt ::= `if (` *expr* `)` *stmt* | `if (` *expr* `)` *stmt* `else` *stmt* | `while (` *expr* `)` *stmt* | `S`

Give an unambiguous grammar that defines all of the above statements and correctly handles the dangling else problem in the grammar. (Hint: introduce multiple productions for statements and divide the productions into two categories: those that might end in an `if` statement with no else clause – a "short if" – and those that definitely do not.)

Question 6. (20 points) Grammars and LR parsing.

Consider the following grammar

s ::= *expr* $
expr ::= a
| a *subs*
subs ::= [*expr*]
| [*expr*] *subs*

(In the first production, $ represents the end of file)

a) (3 points) What are the terminals and non-terminals of this grammar?

Terminals:

Non-terminals:

b) (3 points) Describe in English the set of strings generated by this grammar

(continued next page)

Question 6. (Cont) LR parsing.

(Grammar repeated for reference)

s ::= *expr* $
expr ::= a | a *subs*
subs ::= [*expr*] | [*expr*] *subs*

d) (12 points) Construct (draw) the LR(0) state machine for this grammar. You do **not** need to write out the parser GOTO and ACTION tables, or compute FIRST and FOLLOW sets. Just draw the state machine. Be sure to show the sets of items in each state.

e) (2 points) Is this grammar LR(0)? Why or why not?

Question 7. (18 points) x86 hacking.

Consider the following C main program and function definition:

```
int avg(int a, int b) {
   int ans;
   if (a == b) {
      return a/2;
   } else {
      ans = (a + b) / 2;
      return ans;
   }
}

void main() {
   int x;
   x = avg(17,42);
}
```

(a) (5 points) Draw a picture showing the layouts of the stack frames for methods `avg` and `main`. This should show the stack layout immediately after the prologue code of each function is executed, just before execution of the first statement of the function body (i.e., after the stack frame has been allocated). Be sure to show where registers `ebp` and `esp` point, and the location and offsets from `ebp` of each parameter and local variable.

Question 7. (cont)

(b) (13 points) Translate both functions `avg` and `main` into x86 assembly language. You do not need to slavishly imitate the code shapes described in class – straightforward x86 code is fine as long as it is correct, uses the registers properly, and obeys the x86 C language conventions for stack frame layout, function calls, and so on. It will help us grade your answer if you include the source code as comments near the corresponding x86 instructions. The C code is repeated here to save some page flipping.

```
int avg(int a, int b) {
   int ans;
   if (a == b) {
      return a/2;
   } else {
      ans = (a + b) / 2;
      return ans;
   }
}
```

```
void main() {
   int x;
   x = avg(17,42);
}
```

CSE 582 Autumn 2002 Exam 11/26/02

Question 8. (10 points) Code Shape

Several languages in the ALGOL family had a loop statement designed strictly for counting loops. The syntax for a specific example is:

`for` *var* `:=` *expr1* `to` *expr2* `by` *expr3* `do` *statement*

The semantics of the `for` statement are to repeatedly execute the *statement* that is the loop body with *var* taking on values starting at *expr1*, continuing while the value in *var* is less than or equal to *expr2*, and incrementing *var* by *expr3* after each execution of the loop body. All three expressions (*expr1*, *expr2*, and *expr3*) are evaluated **only once before** the first iteration of the loop. If *expr1* is initially greater than *expr2* then the loop body is not executed. The final `by` *expr3* clause is optional; if it is omitted, `by 1` is assumed instead.

Describe the code shape that could be used to implement this statement on the x86 processor, in the same style that we've used to show code shapes for other programming language constructs. Include the instructions and labels needed to implement the loop and be sure it is clear where the compiled code for the various expressions and the loop body would appear. If you introduce any temporary variables, be sure it is clear where they are stored (where in registers and/or memory).